LAMBDA EXPRESSIONS

Java 8!

Reacting to a Button Press

- A lambda expression to react to a button press

```
button.setAction(event -> System.out.println("Thanks for clicking!"));
```

lambda expression

- Do not call immediately, but only when button is pressed

```
button.setAction(event -> System.out.println("Thanks for clicking!"));
```

parameter(s) function body

- Lambda expression: Anonymous function
  - Parameters, function body, free variables (none in this example)
  - Can be stored like data for later execution

Lambda Expressions

- An expression with parameter variables (and free variables)
  - (int x) -> { return 2 * x + 1; } or x -> 2*x + 1
- Why lambda?
  - In the 1930s Alonzo Church tried to formalize computability using function abstraction, function application, and variable binding
  - Principia Mathematica (Whitehead, Russel, 1910-13) used $2\hat{x}+1$ to denote function f with f(x) = 2x+1. Alonzo Church wanted the notation $\hat{x}.2x+1$, the typesetter could only do ^x.2x+1 (uppercase lambda), another typesetter changed this to λ x.2x+1 (lowercase lambda)
  - λ x.2x+1 is called a lambda abstraction for the function f(x) = 2x+1
- Function application in lambda calculus
  - (λ x . 2x+1) 3 = 2*3+1 = 7
  - (λ op .(λ x .op x x)) (+) 3 = (λ x .(+) x x) 3 = (+) 3 3 = 6

Functions as Values / Code as Data

- Functions as values
  - Store in variables, pass as arguments to other functions, return from functions
- Remember C function pointers
  - Function pointer is memory address of machine code of a function
- Why functions as values?
  - Callbacks in GUIs (button clicked)
  - Separating iteration and operation on elements (filter elements from list)
  - Comparator functions for sorting (case-insensitive string sorting)
  - Parallel operations on big data (function describes operation on each element)
  - Allows for very general and concise code

Things – Actions

Things

Nouns

Data

Actions

Verbs

Procedures

Knock down this wall...

Brian Harvey, MIT CS 61A, The Structure and Interpretation of Computer Programs, Lecture 3: Functions of Functions

Lambda Expressions in Java

(int x, int y) -> { System.out.println(x + y); }
(x, y) -> System.out.println(x + y)
(int x, int y) -> { return x+y; }
(x, y) -> x + y
x -> 2 * x

- Syntax:
  (formal-parameter-list) -> { expression-or-statements }
- (formal-parameter-list)
  - (int x, int y)
  - (x, y) may omit variable type, will be inferred
  - x single parameter, may omit parentheses
  - ( ) empty parameter list
- {expression-or-statements}
  - { return x+y; } statements must be enclosed in braces
  - x+y single expression or method call, may omit parentheses
  - { System.out.println(x+y); }

Interfaces in Java

- Interfaces specify method signatures without implementing them
  - (Java 8 allows implementing static and default methods)
- The set of method signatures of an interface specify a type
- Interfaces specify what is provided by a component
- Objects can later implement these interfaces

- Example interface

```
interface Animal {
    void eat(String food);
    String speak();
    boolean canFly();
}
```

- Example class implementing interface

```
class Cat implements Animal {
    public void eat(String food) { // eating... }
    public String speak() { return "miau"; }
    public boolean canFly() { return false; };
}
```

Functional Interfaces to Specify Lambda Expressions

- Lambda expressions do not have explicit type
  - Compiler infers target type from context
  - Target type is type of object to which lambda expression is bound
  - Target type is a functional interface
- A functional interface has a single abstract method
- Example

```
interface DoubleToDouble {
    double apply(double x);   // takes a double, returns a double
}
```

- Assigning a lambda expression to this type

```
DoubleToDouble square = x -> x * x;
```

- Applying the lambda expression to 3

```
double d = square.apply(3);
```

Manipulating Sentences

```
interface StringToBool { // functional interface
    boolean apply(String s);   // takes a string, returns a boolean
}
String keep(StringToBool predicate, String s) { // keep words that match predicate
    String[] words = s.split("[ ,.!;]+"); // split the sentence at spaces, commas, etc.
    StringBuilder sb = new StringBuilder();
    for (String w : words) {
        if (predicate.apply(w)) { // append w if predicate is true for the w
            sb.append(w); sb.append(' ');
        }
    }
    return sb.toString();
}
```

Manipulating Sentences

```
String t = keep(s -> s.contains("s"), "this is how you see it");
// returns "this is see "

String t = keep(s -> s.charAt(0) == s.charAt(s.length() - 1), "radar rain tilt door");
// returns "radar tilt "

String t = keep(s -> isNumber(s), "the 1 after 2 is 3!");
// returns "1 2 3 "

boolean isNumber(String word) {
    for (int i = 0; i < word.length(); i++)
        if (!Character.isDigit(word.charAt(i))) return false;
    return true;
}
```

Manipulating Sentences

```
Strint t = keep(s -> isPronoun(s), "this is how you see it");
// returns "you it "

String[] pronouns = { "I", "me", "you", "he", "she", "it", "him", "her", "we", "us" };

boolean isPronoun(String word) {
    for (String p : pronouns) {
        if (p.equals(word)) return true;
    }
    return false;
}
```

Method and Constructor References

- Instead of writing
  String t = keep(s -> isPronoun(s), "this is how you see it");
  - Lambda expression only forwards argument to desired function

- better write
  String t = keep(this::isPronoun, "this is how you see it");
  - A reference to an instance method

- Method references to static methods (ClassName::staticMethod)
  - DoubleToDouble f = Math::sin; // static method sin
  - double x = f.apply(1.1);

Method and Constructor References

- Method references to instance methods (ClassName::instMethod)
  - interface StringToInt {
    int apply(String s); // take String, return int
    }
  - StringToInt len = String::length; // instance method of class String
  - int n = len.apply("hello"); // object passed as first argument

- Constructor references (ClassName::new)
  - interface IntToInteger {
    Integer apply(int i); // take (primitive) int, return Integer object
    }
  - IntToInteger iti = Integer::new; // constructor of class Integer
  - Integer k = iti.apply(123);

Variables from Enclosing Scope, Closure

- Lambda expression can capture the value of a variable in the enclosing scope

```
final int minLength = 4; // final (constant) variable in enclosing scope
String t = keep(s -> s.length() >= minLength, "this is how you see it");
// returns "this "
```

- Free variables: Not parameters and not defined in lambda body
- Can only capture (effectively) final local variables and any object and class variables
  - Effectively final: Variable that does not change (after initialization)
- Lambda expressions do not introduce a new level of scoping

Effectively Final Local Variables, Closure

- Effectively final local variables

```
String f(String t) {
    int minLength = 4; // effectively final (not modified after initialization)
    t = keep(s -> s.length() >= minLength, t);
    // minLength++;    // uncommenting this yields a compiler error
    return t;
}
```

- A closure is the combination of
  - Parameter variables (s in example above)
  - Function body (code in {...} in example above)
  - Values for free variables (4 in example above)

Effectively Final Local Variables, Closure

- Return a function from a function

```
StringToBool makePredicate(int minLength) {
    return s -> s.length() >= minLength;
}
```

- Returned StringToBool-function can be used after makePredicate has completed!

```
StringToBool f = makePredicate(4);
System.out.println(f.apply("1234")); // returns true
System.out.println(f.apply("123")); // returns false
```

Processing an Array of Persons

```
class Person {
    String name;
    int age;
    String emailAddress;
    Person(String name, int age, String emailAddress) { // constructor
        this.name = name;
        this.age = age;
        this.emailAddress = emailAddress;
    }
    public String toString() {
        return "Person(" + name + ", " + age + ", " + emailAddress + ")";
    }
}
```

Processing an Array of Persons

- An array of persons

```
Person[] ps = {
    new Person("Karl", 23, "karl@abc.de"),
    new Person("Franz", 29, "franz@def.de"),
    new Person("Alice", 19, "alice@jkl.de"),
    new Person("Fritz", 25, "fritz@ghi.de"),
    new Person("Alina", 35, "alina@abc.de")
};
// print the array
for (Person p : ps) {
    System.out.println(p.toString());
}
```

- Output

```
Person(Karl, 23, karl@abc.de)
Person(Franz, 29, franz@def.de)
Person(Alice, 19, alice@jkl.de)
Person(Fritz, 25, fritz@ghi.de)
Person(Alina, 35, alina@abc.de)
```